ICASE

CAPABILITIES FOR HIGH LEVEL LANGUAGES

Martin S. McKendry

and

Roy H. Campbell

Report No. 80-32

November 24, 1980

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia

Operated by the

UNIVERSITIES SPACE USRA RESEARCH ASSOCIATION

CAPABILITIES FOR HIGH LEVEL LANGUAGES

Martin S. McKendry
University of Illinois at Urbana-Champaign

and

Roy H. Campbell
University of Illinois at Urbana-Champaign

ABSTRACT

Language capabilities are a mechanism for dynamic resource management and protection. They are implemented as pointers whose use is constrained by compile-time checks. Capability chains facilitate hierarchical resource management and efficient access to dynamically managed data structures. The use of capabilities is illustrated with an example message passing system. The efficiency and generality of language capabilities suggests their application to the construction and capability-based, level structured operating systems.

Work was supported in part by NASA Project No. NSG-1471 and NASA Contract No. NAS1-14472 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1 Introduction.

Operating systems use protection mechanisms to ensure consistent environments for concurrent users, to reduce error propagation, and to provide user and system components with security against illegal interference. Hardware supports protection mechanisms such as paging, segmentation, and capabilities [5, 7], while software provides type checking, scoping restrictions and encapsulation. The software mechanisms do not have the generality of hardware protection mechanisms, however, since they are only able to express static protection schemes. Operating system languages need dynamic protection mechanisms that are as flexible as hardware capabilities [1,10]. In this paper, we propose a capability based language feature as an extension to an existing dynamic access mechanism. The feature depends on compile-time checking of access rights and run-time checking of pointer validity.

Existing protection models consider access by subjects to objects. From a practical viewpoint, an operating system is a set of concurrent processes that share a common pool of resources such as buffers, memory segments, and programmed services. In discussing language protection, therefore, we are interested in access by processes to resources (which we will call *data items* or *items*).

Many high level languages, such as Pascal, use strongly typed pointers to access dynamically allocated variables [8]. With *language capabilities*, the notion of pointer typing is extended to include access rights -- the ways in which a pointer may be used. By constraining the modification and copying of capabilities to a specific program module (called a *manager*), language capabilities achieve the same protection, with the same generality, as is achieved by hardware capabilities. However, unlike hardware capabilities,

where access checking must be dynamic, use of language capabilities can be checked statically, thus eliminating a major overhead of many existing hardware capability implementations.

Checking for correct capability use is performed statically by a compiler, but dynamic protection is programmed in much the same way as management of dynamic data structures. The program components that distribute access to dynamically created data items can also, with minor changes, protect those same items. Furthermore, because protection is associated with access, different access rights may be given processes for the same item -- a record can be read-only for one process and read-write for another.

For convenience, language capabilities are described as an extension to Path Pascal [3]. Path Pascal is an object oriented language in the style of Simula [4], Concurrent Pascal [2], CLU [15], and Ada [22]. An object, which is declared as a variable or a type, encapsulates data that can be accessed only through entry procedures and an initialization block. Synchronization is specified by a path expression. The features of Pascal that are critical to our implementation are pointer access to dynamically allocated variables and strong type checking of procedure parameters. Some form of encapsulation is also required. Our examples make extensive use of objects, both as protected items and as managers.

Capability types are declared within their managers, and their automatic export enables processes to declare capability variables. Inside managers, capabilities can be used as if they are pointers, while outside their managers they can be used only in accordance with their access rights.

1.1 Capabilities for Buffers.

We introduce language capabilities with an example buffer manager. It will be used as a building block for our main example, the port mechanism. The buffer manager distributes access to buffers, protecting them by insuring that a process can access those buffers it has been allocated and no others. Access must, therefore, be revoked when a buffer is returned.

The buffer type is a Pascal record:

```
TYPE buffer-type = RECORD
                   size: 0..max-buff;
                   data: ARRAY [0..max-buff] OF CHAR;
                   qptr: ^buffer-type;          (* for free list *)
                   channel: channel-id;         (* used in port mechanism *)
                   END
```

In standard Pascal, a pointer would be used to access buffers. Protected access requires that a pointer be declared:

```
TYPE buffer_cap = CAPABILITY buffer_type; read, write END
```

The capability type declaration specifies the item to which capabilities may be bound and their access rights (the ways in which the items can be accessed). Processes declaring buffer capabilities can use them to read and write buffers. Capabilities can only be assigned new values within their managers. Thus, a process must call the buffer manager to get access to a buffer and, once the buffer is returned, access is no longer possible -- attempts at access result in run time errors (nil pointer references). The buffer manager, which contains the capability declaration, is shown below:

```
TYPE buffer_manager_type = OBJECT
   PATH 1:(get, release),                          (* mutual exclusion *)
        no_of_buffers: (release; get) END;         (* resource limitation *)

   TYPE buffer_cap = CAPABILITY buffer_type; read, write END;

   VAR  headfl     : buffer_cap;                   (* head of free list *)
        b_c        : buffer_cap;                   (* temporary *)
        i          : integer;                      (* temporary *)

   ENTRY PROCEDURE get (VAR b: buffer_cap);    (* set b pointing to a buffer *)
      b := headfl;                           (* remove a buffer from free list *)
      headfl := headfl^.qptr;
      b^.qptr := NIL;

   ENTRY PROCEDURE release (VAR b: buffer_cap); (* accept a returned buffer *)
      b^.qptr := headfl;                        (* place buffer on free list *)
      headfl := b;
      b := NIL;                                 (* revoke capability b *)

   INIT;                                        (* initialization of manager *)
      NEW (headfl);                             (* create buffers *)
      headfl^.qptr := NIL;
      for i := 2 to no_of_buffers do
         NEW (b_c);
         release (b_c);
END; (* buffer_manager *)
```

Example 1: Buffer Capability Manager.

Code that can access the buffer manager can declare a buffer capability, which is then passed to the buffer manager to acquire buffers (buffer_manager is assumed to be of buffer_manager_type):

```
VAR b_cap: buffer_cap;

buffer_manager.get (b_cap);
```

Any field can be accessed by using the capability as a pointer:

```
b_cap^.data [4] := 'x';
```

This example raises several issues. These include control of buffer-type instantiation, control of access to the buffer manager, and the use of capabilities as parameters. These issues, and the issues of capability chains,

revocation, and implementation are addressed in Chapter 2. Chapter 3 presents the port mechanism, an example of a protected message passing system. Chapter 4 reviews previous work on language protection and capabilities, then Chapter 5 discusses several protection issues. Finally, Chapter 6 comments on the use of language capabilities for operating system construction.

2 Language Capabilities.

Five characteristics distinguish language capabilities from ordinary pointers. These characteristics combine to provide strong protection and revocation abilities:

1) Access rights are specified when capability types are declared.
2) The type of a capability is automatically exported from the object or procedure in which it is declared. This object or procedure is called the manager of that capability. It controls access to the protected item.
3) Outside managers, capabilities can only be used in accordance with access rights.
4) Outside managers, capabilities cannot be copied or assigned.
5) Capabilities may form chains that are automatically dereferenced.

For simple (non-object) data items the only possible access rights are read and write, but for Path Pascal objects the possible access rights are the entry procedures of the object (e.g., a capability for the buffer_manager_type could have 'get' and 'release' as rights). Export of program types is used in several languages (e.g., Modula [23]) to construct variables whose internal structure is inaccessable to all program components except the manager for that type. We use export to construct variables (capabilities) whose internal structure is visible and usable, but which cannot be copied outside their

managers.

2.1 Chains.

Capability chains occur whenever capabilities for capabilities are defined. If any capability on a chain is dereferenced outside its manager, the rest of the chain is followed to the final item. Thus, levels of management may be made transparent to processes. Chains are used to construct hierarchical protection schemes and as a means of sharing access to protected variables without copying capabilities.

When a chain is used, the item being protected does not change even though a level of management is added. Consequently, a distinction is needed between the item protected by a manager (note that a manager which declares a capability for a capability is protecting capabilities), and the item type to which a chain is ultimately bound. The former type is known as the managed type of a capability. It can be any valid type, including another capability. The latter type is called the item type of any capability on the chain.

Within its manager, the rule that a capability is used as a pointer means that a single dereference produces a variable of the managed type (not the item type). Only one link of the chain is followed, a second dereference being needed to follow the rest of the chain. This property is used in the port mechanism below. It means that within a manager a clear distinction is seen between manipulation of the managed type and manipulation of the item type.

The access rights of a chain are those of its first link. Thus, the access rights of a capability apply to the item type of the capability. Capa-

bilities for capabilities may have only subsets of the rights of their managed types. When a capability is bound directly to an item, the capability may have as rights any of the valid operations on the item. This stops a manager from distributing greater rights to an item than it itself has. Consider Example 2:

```
TYPE c1 = CAPABILITY buffer_type; read, write END;
        :
        :
TYPE c2 = CAPABILITY c1; read, write END;
        :
        :
TYPE c3 = CAPABILITY c2; read END;

VAR v1: c1;
    v2: c2;
    v3: c3;
```

This enables a chain to be built:

v3 --> v2 --> v1 --> a buffer variable

Example 2: Capability Chains.

In this example c1, c2, and c3 are considered to have distinct managers. Thus, a capability of type c3 (such as v3) represents the third level of management. The item type of c1, c2, and c3 is the buffer_type, while the managed types are buffer_type, c1, and c2 respectively. If v1, v2, or v3 is dereferenced outside its manager, the chain will be followed to the final buffer instance.

2.2 Revocation.

A manager controls the number of capabilities bound to an item. Consequently, normal revocation can be accomplished by setting capabilities to 'nil' when they are returned by a process, thus removing the access path from

process to item (e.g. 'release' in the buffer manager).

A simple form of preemptive revocation can be implemented with an extra link on the chain [23]. The capability manager distributes not the capability itself, but a capability for it, so that the original capability is retained by the manager and can be set to 'nil' when preemptive revocation is required. As a result of automatic chain dereferencing, the extra level of indirection is transparent to processes.

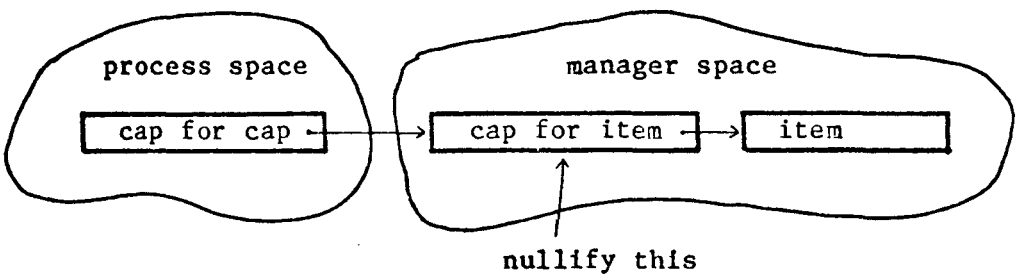

Figure 1: Chain-based Preemptive Revocation.

A more sophisticated form of preemptive revocation, a method that requires processes using revokable capabilities to be aware of revocability, is to have the manager retain a pointer into a process's space, thereby enabling the manager to revoke a capability directly (Figure 2). This requires the user process to pass the manager a pointer to a capability rather than the capability itself. The manager then makes a copy of the pointer before issuing the capability. The disadvantage of user-awareness in this scheme is offset by the advantage that no non-reusable 'slots' are created, as happens with the first scheme.

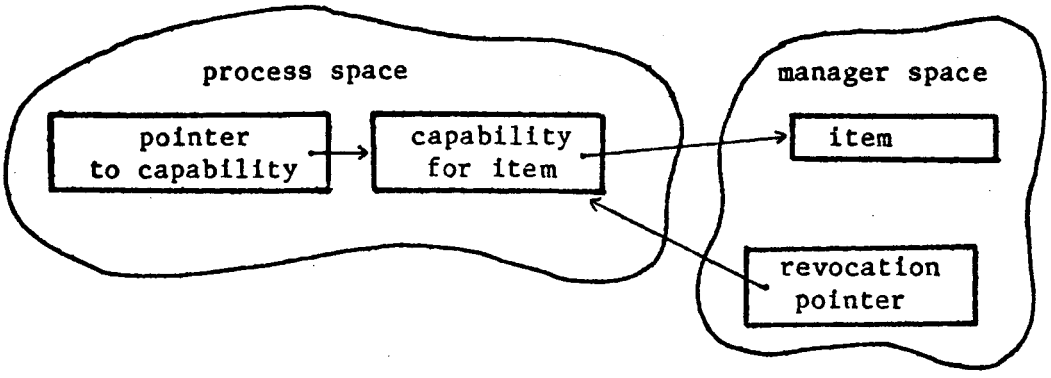

Figure 2: Pointer-based Preemptive Revocation.

Revocation of a capability for an item containing code (an object) requires that the revoker ensure that no process is active within the item before considering revocation to be complete. Path expressions and procedure entry counts are two mechanisms that can be used to do this.

2.3 Capability Parameters.

Use of capabilities as procedure parameters is limited to calls by reference (by VAR in Pascal). Calls by reference create a pointer (invisible to the programmer) to the capability being passed, maintaining the policy that managers have total control over the issuing and copying of capabilities.

To reduce rights, capabilities require type coercion when they are passed as parameters outside a manager. Coercion is a bracketing construct with the new type of the capability enclosing the actual capability. The rights of the new type must be a subset of the rights of the capability being coerced. The type used to coerce a capability need not be known to the manager for that capability, but must be known to both caller and callee. Consider Example 3:

```
TYPE b_cap       = CAPABILITY buffer; read, write END;
     b_read_only = CAPABILITY buffer; read END;

PROCEDURE read_buffer (VAR b: b_read_only);
   :
   END;

VAR local_b: b_cap;

read_buffer ( b_read_only (local_b) );  (* example of legal coercion *)
```

Example 3: Capability Type Coercion.

Example 3 shows two capabilities that have the same item and managed type: a buffer. Since b_read_only has a subset of the access rights of b_cap, a variable of type b_cap can be coerced to type b_read_only in the call on read_buffer. While the containing process has read and write access through local_b, read_buffer can only read the item to which local_b is bound.

2.4 Scoping Constraints.

We use the Restrict pseudo-statement [1] to control instantiation of protected types. The declaration, which represents function rather than a particular syntax, is clearly more efficient than programmed checks. Restrict limits instantiation (NEW in Pascal) and elimination (DISPOSE) of protected items to a managing object or procedure, without constraining access by processes to the internal structure of protected types. An example, used in the port mechanism, is:

RESTRICT buffer-type TO buffer-manager

2.5 Implementation.

No additional run-time checking mechanisms are needed to implement language capabilities. All changes are to the compiler, which must recognize capability declarations and the Restrict statement, and enforce correct use. Inside its manager, a capability may be used as a pointer. Outside its manager, a capability for an object cannot be used except to call the entry procedures specified in the capability's access rights. Capabilities for simple data items may be used in any expression if the read right is set; if the write right is set the capability may be used on the left of assignment statements. The usual strict typing, augmented with the coercion described above, is used to control capability parameter passing.

Structures, such as records and arrays, that contain capabilities cannot be copied. To avoid the 'forging' of capabilities, capabilities cannot be placed in variant fields of records.

On procedure entry, all capabilities declared locally must be initialized to nil to avoid 'dangling' pointer references. Concurrency considerations dictate that capability chains must be completely dereferenced each time they are used — some sub-expression optimizations are not possible. In particular, the 'with' statement may not circumvent chain dereferencing, but instead must follow the entire chain on each access. Mutually recursive management of object capabilities can lead to an infinite chain of dereferencing that must be flagged as a compile-time error.

3 Message Communication Using Capabilities.

Our main example shows the capabilities and data types used to implement a 'port' mechanism. The port mechanism illustrates capability features required to build a protected file system [16]. It demonstrates simple revocation, the granting of different access rights to the same item, and hierarchical management with capability chains. Capabilities for both simple data items and Path Pascal objects are used.

The example is based on a 'canonical' message passing system (described in [14]) in which processes communicate through ports (Figure 3). Three objects are used to build the mechanism: the buffer manager (shown above), the port itself, and the port manager. Each process is assigned a port object by the port manager. Requests for empty buffers and for buffer passing are then handled by the process's port. The ports use a common pool of buffers maintained by the buffer manager to satisfy requests for buffer acquisition and release. A process can access only the port allocated to it by the port manager and the buffers passed to it by its port.

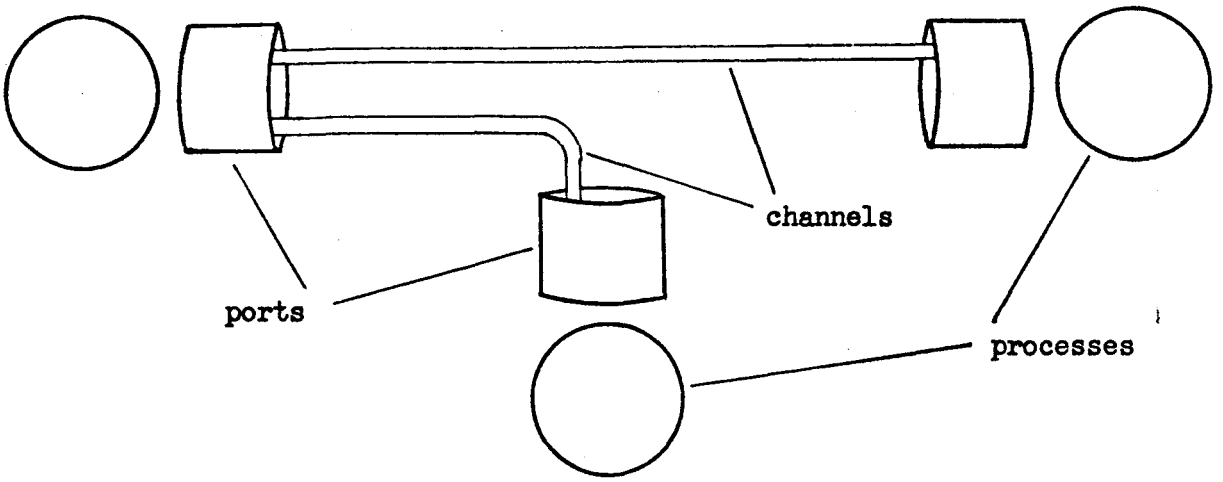

Figure 3: Representation of Message System.

Ports are subdivided into channels. Incoming buffers are held on a single queue in the port. A process can tell from which channel a buffer came by the 'channel' field in the buffer — the port does not supply the identity of the process that sent the buffer.

3.1 Process Identities.

To protect ports against illegal access some means of identifying processes is needed. This is implemented by a 'process id manager' that issues identities to processes. Process identifiers are accessed through a process identifier capability:

```
TYPE process_id_cap = CAPABILITY integer; read END
```

A set of these capabilities is generated by the 'basic' operating system, which then issues them to processes as the processes are instantiated:

```
PROCESS any_process (VAR id: process_id_cap);
... code ...
END; (* any_process *)
```

Because they are accessed through a read-only capability, process identifiers are unique, unforgeable, and indestructible. The strong typing (name equivalence) of parameter passing ensures that, when passed as parameters, process identifier capabilities correctly identify the calling process. The identity of a process is found by dereferencing its process_id capability.

3.2 Ports.

Port objects are at the 'outer edge' of a management hierarchy in which levels are strictly separated — the buffer manager is independent of the additional management implemented in the ports, and processes are independent of the buffer manager. To support this separation, ports use explicit queueing records. The bindings from processes to buffers (a chain of 2 capabilities) and the queues of buffers are depicted in the Appendix.

The appendix also contains the coding of the port mechanism in Path Pascal. The process associated with a port uses the entry procedures 'user_get' and 'user_release' to acquire buffers and release them. The port gets buffers from the common pool maintained by the buffer manager. User processes call 'send' and 'receive' to communicate with other processes. The procedure 'supply' implements inter-port communication by accepting a buffer sent by a source port and enqueueing it at the destination port. The port manager uses configuration routines 'init_buffer_manager' and 'set_channel' to arrange port access to the buffer manager and port access to other ports. A

path expression in the port controls both mutual exclusion for the queue data structure and blocking when executions of 'receive' are attempted with an empty buffer queue. Instantiation of managed items (buffers, the buffer manager, and ports) is constrained by Restrict statements.

To protect it from direct access by processes, ports access the buffer_manager through a pointer. A restrict statement authorizes the port manager to create buffer managers; distribution of pointers to the buffer manager is limited to ports. Thus, the buffer manager is common to all ports, yet cannot be accessed by other program elements.

When a buffer is first requested from the buffer_manager, the port attaches a buff_cap to it as a 'tag' that the port can manipulate. This is removed when the buffer is returned to the buffer_manager. A user process uses user_buff_caps to access buffers. These are set to nil when not actually bound to a buffer.

To communicate with other processes through the port mechanism a process first declares a port capability and calls the port manager to have it bound ('id' is a process identifier capability):

```
VAR my_port: user_port_cap;          (* declare port capability *)
port_manager.issue_user_port (id, my_port); (* initialize it *)
```

The capability 'my_port' is then used to acquire, pass, and release buffers. Variables of type 'user_buff_cap' contain capabilities for buffers:

```
VAR b: user_buff_cap;          (* declare a capability for a buffer *)
my_port^.user_get (b);         (* request an empty buffer *)
b^.data := 'a message';        (* place information in the buffer *)
my_port^.send (b,4);           (* send buffer via channel 4 *)
```

```
my_port^.receive (b);          (* receive an incoming buffer --
                                  or block until there is one *)
```

The only global, statically instantiated portion of the message system is the port manager. All other structures are instantiated dynamically -- capabilities are used to grant access on a 'need to know' basis. At no time can a process or port access any buffers, ports, or entry procedures other than those to which access has been explicitly granted.

4 Capabilities and Dynamic Protection.

Capabilities, proposed by Dennis and Van Horn [5] and later refined by Fabry [7], were originally a low-level, hardware-oriented concept. A capability was a hardware-defined name or address for a data item, constraints on which were used to achieve highly protected systems. Several capability machines have been implemented [6,18]. These interpret access rights and unique item addresses during execution using microcode and associative memories. Their capabilities are either stored in special capability segments, or are distinguished from data by tag bits that specify access rights. Capability operating systems for traditional machine architectures have also been implemented [19,24]. These use access interpretation (or 'soft' addressing mechanisms) in a secure kernel to enforce protection.

Jones and Liskov [10,11] initially suggested implementing capabilities for languages, but did not consider managers specifically. Instead, any process could create an item and pass it with reduced rights to other processes or procedures. Their coercion was automatic -- we consider it preferable that an explicit language construct be used to coerce types. However, their use of compile-time checking and static association of rights with capabilities is

similar to our own, and our mechanism can be specified in terms of their model [17].

Silberschatz, Kieburtz and Bernstein [21] approached capability protection motivated by nested monitor problems that arise in Concurrent Pascal. They introduced a manager construct to the language, and the standard procedures 'bind' and 'release' to control the number of bindings of capabilities to items with a hidden count. Because of this, their mechanism requires greater implementation effort than ours, and is not as efficient or as flexible. The use of an explicit manager construct prohibits construction of hierarchical management schemes. Further run-time support and an additional language feature are needed to implement preemptive revocation [12].

McGraw and Andrews [16] proposed a capability scheme for Concurrent Pascal in which access rights are checked dynamically. While much of the functionality of their scheme is similar to ours, we do not feel that the overheads of dynamic checking are warranted. Their protected variables [1] associate protection with a variable rather than with access to it. This makes an unfortunate distinction between protection of monitors and protection of simple variables, and reduces the orthogonality of protection and synchronization.

In previous work on language capabilities, the synchronized program object has been emphasized as the unit of protection. Pointers, however, can be used for all data structures. Since pointers can readily be modified to build capabilities, we consider the emphasis on object protection to be unnecessary. The generality of our scheme simplifies implementation, facilitates construction of capability based operating systems, and separates the orthogonal concepts of synchronization and protection.

Ada [22], a newer language than those discussed above, has protection facilities similar to those of Modula [23]. Private types are exported from packages, so that other program components can possess variables of those types. In direct contrast to our scheme, however, the structure of private types is inaccessible outside packages (managers), and they can be freely copied by any program component. Thus, a port mechanism (or file system) that relies on protected pointers for access to buffers and port (or file) objects cannot be built. Instead, protected variables must be passed to procedures in packages to access internal fields or entry procedures. This incurs considerable overhead.

5 Discussion.

5.1 The 'Copy Right'.

Implementation of chains has allowed us to dispense with the 'copy right' provided in many schemes [16,13]. This simplifies revocation and coercion, and there is no longer any need to control capability assignments outside managers. The need for a 'nullify' right is also eliminated [1]. This enhances reliability, since capabilities cannot be accidentally nullified thereby 'losing' resources. Similarly, it is not possible for a process to destroy a capability by assigning it a legitimately held 'copyable' capability.

5.2 Impersonation.

If processes can pass each other access to process identifiers or port capabilities, protection violations can occur. This 'impersonation' is a general problem that we do not intend to prohibit at the language level.

Instead, we direct attention to the structure of the overall system. Consider the example configuration in Figure 4:

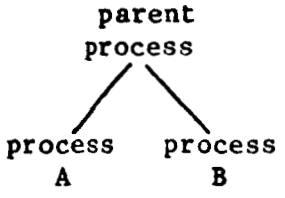

Figure 4.

Our contention is that, in a well structured system, processes A and B should communicate only by passing capabilities through some common ancestor (the parent process). The conversation between A and B can then be monitored at that ancestor, thus building a 'firewall' that disallows the parameter passing needed for impersonation. If the parent does not have this control over A and B they should be viewed as a single process. Language mechanisms that provide the scoping control needed to implement this level of static protection already exist [22].

6 Capability Operating Systems.

One of the main advantages of the generality of our mechanism is its utility for operating system construction. An operating system can use language capabilities to control access by processes to protected structures and services. Capabilities can also be used to control access between levels in level structured operating systems. A scheme for implementing logical I/O is depicted in Figure 5.

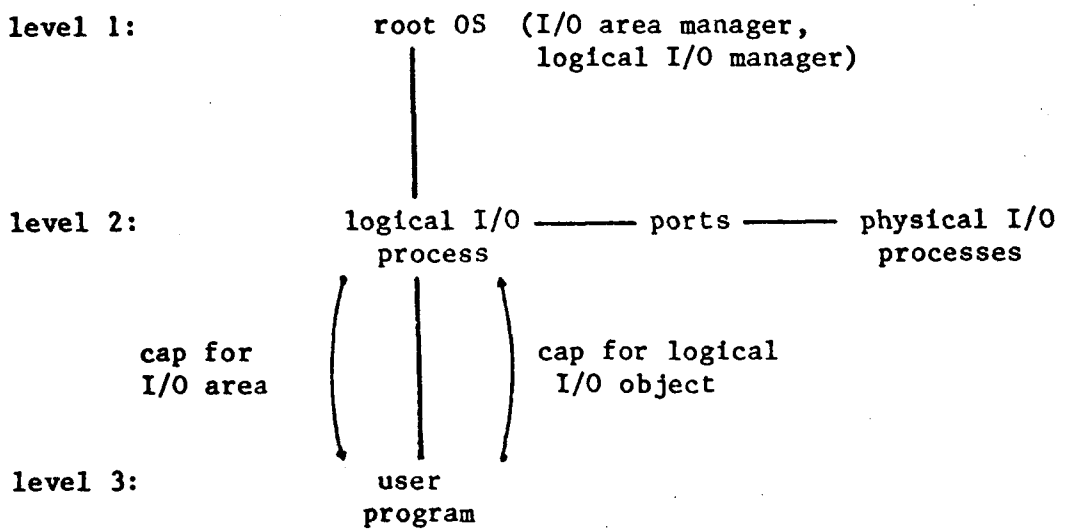

Figure 5.

In this scheme the logical I/O process is responsible for translating device-oriented physical I/O to the logical I/O required by user processes. When the user is loaded, the root operating system issues it a capability for logical I/O. When an I/O request is made, a capability for logical I/O is passed as a parameter to the logical I/O object. The I/O transfer is then made without any possibility of the I/O object interfering in other memory space. The I/O area capability is automatically revoked when the transfer is complete. Operating system modules are thus prevented from unauthorized access to user memory space, and user programs are prevented from making direct access to system memory space.

Using this scheme the loading of a user program is actually a strongly typed binding. In [17], specification is made of the dynamic binding requirements and language mechanisms for control of the processor state transitions needed to execute user programs and recognize their requests for service.

7 Summary.

We have described language capabilities, a scheme for dynamic access control that is efficient, easily implemented, and that leads to clearly visible program protection structures. Language capabilities are based on a type extension of pointers. Capabilities and capability chains are consistent with existing data types and require no additional run time support mechanisms. Capabilities are declared in managers and their types are exported so that other program components can declare capability variables. They can be used to construct arbitrarily nested management schemes for both encapsulated and simple data structures.

An example message passing facility has been used to illustrate the use of capabilities. The facility includes unforgeable process identities, a common buffer pool, a port mechanism for buffer passing and management, and direct access to buffer records once they are issued to processes. The features of capabilities used to build the port manager are the same as those required to build a 'file' mechanism.

Finally, we have indicated the utility of language capabilities for operating system construction. Checking of capability access rights can be implemented at run-time by a tagged architecture. Alternatively, a reliable compiler can check access rights at compile time. Because of increased efficiency that results from this static checking, we predict that language capabilities will play a major role in future operating system implementation languages.

8 Acknowledgements.

We would like to thank Paul Richards for his participation in discussions of this scheme and Rob Kolstad for his editorial comments.

9 References.

[1] G. R. Andrews and J. R. McGraw, "Language Features for Process Interaction," Proc. ACM Conf. on Language Design for Reliable Software, Sigplan Notices, Vol. 12, No. 3, pp. 114-127, Mar. 1977.

[2] P. Brinch Hansen, "The Programming Language Concurrent Pascal," IEEE Trans. on SE., Vol. SE-1, pp. 199-207, June 1975.

[3] R. H. Campbell and R. B. Kolstad, "An Overview of Path Pascal's Design," Sigplan Notices, Vol. 15, No. 9, pp. 13-14, Sept. 1980.

[4] O. J. Dahl and C. A. R. Hoare, "Hierarchical Program Structures," in Structured Programming, Academic Press, London, 1972.

[5] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," CACM, Vol. 9, No. 3, pp. 143-155, Mar. 1966.

[6] D. M. England, "Capability Concept Mechanisms and Structure in System 250," Colloques IRIA Intl. Conf. on Protection in Operating Systems, pp. 63-82, Aug. 1974.

[7] R. S. Fabry, "Capability-Based Addressing," CACM, Vol. 17, No. 7, pp. 403-412, July 1974.

[8] K. Jensen and N. Wirth, Pascal User Manual and Report, Springer-Verlag, New York, 1974.

[9] A. K. Jones "Protection in Programming Systems," Ph.D. Thesis, Carnegie Mellon University, 1973.

[10] A. K. Jones and B. H. Liskov, "A Language Extension for Controlling Access to Shared Data," IEEE Trans. on Software Eng. Vol. SE-2, No. 4, Dec. 1976.

[11] A. Jones and B. Liskov, "A Language Extension for Expressing Constraints on Data Access," CACM, Vol. 21, No. 5, pp. 358-367, May 1978.

[12] R. B. Kieburtz and A. Silberschatz, "Capability Managers," IEEE Trans. on Software Eng. Vol. SE-4, No. 6, pp. 467-477, Nov. 1978.

[13] B. W. Lampson, "Protection," Proc. 5th Annual Princeton Conference on Information Sciences and Systems, Dept. of E.E., Princeton Univ. pp. 437-443, Mar. 1971.

[14] H. Lauer and R. Needham, "On the Duality of Operating System Structures," Second International Symposium on Operating Systems, IRIA, Oct. 1978.

[15] B. H. Liskov, et al., "Abstraction Mechanisms in CLU," CACM, Vol. 20, No. 8, pp. 565-576, Aug. 1977.

[16] J. R. McGraw and G. R. Andrews, "Access Control in Parallel Programs," IEEE Trans. on Software Eng., Vol. SE-5, No. 1, pp. 1-9, Jan. 1979.

[17] M. S. McKendry, "Mechanisms for High Level Language Operating Systems," Ph.D. Thesis, University of Illinois at Urbana-Champaign. In preparation.

[18] R. M. Needham and R. D. H. Walker, "Protection and Process Management in The 'CAP' Computer," Colloques IRIA Intl. Conf on Protection in Operating Systems, pp. 155-160, Aug. 1974.

[19] P. G. Neumann, et al., "On the Design of a Provably Secure Operating System," Colloques IRIA Intl. Conf. on Protection in Operating Systems, pp. 161-175, Aug. 1974.

[20] D. D. Redell and R. S. Fabry, "Selective Revocation of Capabilities," Colloques IRIA Intl. Conf. on Protection in Operating Systems, pp. 197-209, Aug. 1974.

[21] A. Silberschatz, et al., "Extending Concurrent Pascal to Allow Dynamic Resource Management," IEEE Trans. on Software Eng., Vol. SE-3, No. 3, pp. 210-217, May 1977.

[22] P. Wegner, Programming with Ada, Prentice-Hall, Englewood Cliffs, NJ., 1980.

[23] N. Wirth, "Modula, A Language for Modular Multiprogramming," Software Practise & Experience, Vol. 7, pp. 3-35, Jan. 1977.

[24] W. Wulf, et al., "HYDRA: The Kernel of a Multiprocessor Operating System," CACM, Vol. 17, No. 6, pp. 337-345, June 1974.

10 Appendix: The Port Mechanism.

10.1 Binding of Capabilities to Buffers.

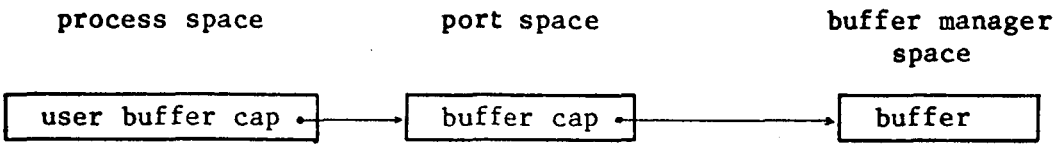

Figure 6 (a): Process Binding to Buffers.

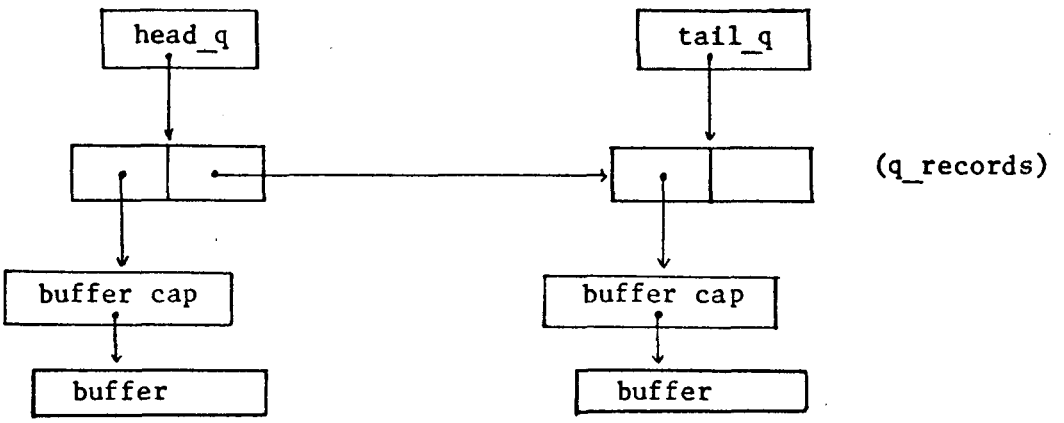

Figure 6 (b): Queuing of Buffers in Ports.

10.2 The Port and Port Manager Objects.

```
RESTRICT buffer_type TO buffer_manager_type;
RESTRICT buffer_manager_type TO port_manager;
RESTRICT port_type TO port_manager;


TYPE port_type = OBJECT
   PATH 1: (send, supply, receive, set_channel),      (* mutual exclusion *)
           (supply; receive) END;                     (* queue constraints *)
```

```
TYPE channel_id    = 0..max_channel;
     to_port_rec   = RECORD
                       to_port    : port_supply_cap;
                       to_channel : channel_id;
                       END;
     user_buffer_cap = CAPABILITY buffer_cap; read, write END;
     q_record      = RECORD
                       buffer     : user_buffer_cap;
                       next       : ^q_record;
                       END;


VAR my_channels : ARRAY [1..max_channels] OF to_port_rec;
    head_q : ^q_record;
    tail_q : ^q_record;
    temp_q : ^q_record;
    buffer_manager: ^buffer_manager_type;

ENTRY PROCEDURE user_get (VAR b: user_buffer_cap);
                                        (* called by a user process to *)
                                        (* get a buffer from common pool *)
                                        (* return b = NIL if there are none *)
   IF b = NIL THEN NEW (b) ELSE error; (* b is still bound *)
   buffer_manager^.get (b^);

ENTRY PROCEDURE user_release (VAR b: user_buffer_cap);
                                        (* called by a user process to *)
                                        (* release buffer to common pool *)
   IF b = NIL THEN error
      ELSE IF b^ = NIL THEN error;
   buffer_manager^.release (b^);
   DISPOSE (b);

ENTRY PROCEDURE send (VAR b: user_buffer_cap;
                          c: channel_id);
                (* called by a user process to *)
                (* send a buffer to port indicated by 'c' *)
IF b = NIL THEN error                           (* no buffer_cap attached *)
   ELSE IF b^ = NIL THEN error;                 (* no buffer attached *)
   b^^.channel := mychannels [c].to_channel;    (* mark channel on buffer *)
   my_channels [c].to_port^.supply (b);         (* send it *)
   b := NIL;
```

```
   ENTRY PROCEDURE supply (VAR b: user_buffer_cap);
                  (* called by another port in 'send'. *)
                  (* accept incoming buffer and enqueue it *)
      IF head_q <> NIL THEN
         NEW (tail_q^.next);
         tail_q := tail_q^.next;
      ELSE
         NEW (head_q);
         tail_q := head_q;
      tail_q^.next := NIL;
      tail_q^.buffer := b;

   ENTRY PROCEDURE receive (VAR b: user_buffer_cap);
                  (* called by user process to get an incoming buffer *)
                  (* remove buffer from queue and release it to process *)
      IF b <> NIL then error;                          (* still bound *)
      b := head_q^.buffer;
      temp_q := head_q;
      head_q := head_q^.next;
      DISPOSE (temp_q);

   ENTRY PROCEDURE init_buffer_manager (b_m: ^buffer_manager_type);
                  (* called by port_manager in initialization sequence *)
                  (* establish pointer to buffer manager *)
      buffer_manager := b_m;

   ENTRY PROCEDURE set_channel (c: channel_id;          (* 'from' channel *)
                                to_c: channel_id;       (* 'to' channel *)
                                to_p: port_supply_cap); (* 'to' port *)
                  (* called by port_manager to *)
                  (* establish channel mapping *)
      WITH my_channels [c] DO
         to_channel := to_c;
         to_port := to_p;

END; (* port_type *)


VAR port_manager: OBJECT
   PATH issue_user_port, 1: establish_port_mapping END;

   TYPE user_port_cap = CAPABILITY port_type; get, release, send, receive END;
        port_supply_cap = CAPABILITY port; supply END;

   VAR buffer_manager: ^buffer_manager_type;
       port: ARRAY [1..no_of_processes] OF user_port_cap;
       i   : INTEGER;

   ENTRY PROCEDURE issue_user_port (VAR id: process_id_cap;
                                    VAR upc: user_port_cap);
                  (* initialize a user process's port capability *)
       upc := port [id^];
```

```
   ENTRY PROCEDURE establish_port_mapping ( ___ );
       (* check caller's identity in accordance with OS design here *)
       (* use set_channel calls to make a mapping *)

   INIT;  (* port_manager initialization code: establish ports etc. *)

      NEW (buffer_manager);
      FOR i := 1 TO no_of_processes DO
         NEW (port [i]);
         port [i]^.init_buffer_manager (buffer_manager);

END; (* port_manager *)
```